PY-PITS: A Scalable Python Runtime System for the Computation of Partially Idempotent Tasks

Edson Borin*, Caian Benedicto†, Ian L. Rodrigues†, Flávia Pisani*, Martin Tygel‡, and Mauricio Breternitz Jr.§
*Institute of Computing (IC), University of Campinas (UNICAMP) – Campinas, SP, Brazil
Email: {edson, fpisani}@ic.unicamp.br
†Center for Petroleum Studies (CEPETRO), University of Campinas (UNICAMP) – Campinas, SP, Brazil
Email: {caian, ian.liu}@ggaunicamp.com
‡Institute of Mathematics, Statistics and Scientific Computing (IMECC)
University of Campinas (UNICAMP) – Campinas, SP, Brazil
Email: tygel@ime.unicamp.br
§AMD Research – Austin, TX, USA
Email: mauricio.breternitz@amd.com

***Abstract*—The popularization of multi-core architectures and cloud services has allowed users access to high performance computing infrastructures. However, programming for these systems might be cumbersome due to challenges involving system failures, load balancing, and task scheduling. Aiming at solving these problems, we previously introduced SPITS, a programming model and reference architecture for executing bag-of-task applications. In this work, we discuss how this programming model allowed us to design and implement PY-PITS, a simple and effective open source runtime system that is scalable, tolerates faults and allows dynamic provisioning of resources during computation of tasks. We also discuss how PY-PITS can be used to improve utilization of multi-user computational clusters equipped with queues to submit jobs and propose a performance model to aid users to understand when the performance of PY-PITS scales with the number of Workers.**

I. Introduction

With the ever-increasing amount of data being generated every day, it has become apparent that using a single machine is no longer feasible in many real-world applications. While exploiting parallel solutions in multi-core architectures is still of paramount importance, looking at distributed approaches such as using computer clusters and the cloud has become imperative in these situations.

The need for parallelized code has fostered the creation of several Application Programming Interfaces (APIs), programming models, and libraries over the years. To name a few, we have OpenMP [1], OpenCL [2], CUDA [3], TBB [4], and MPI [5]. Still, most of these tools present a series of requirements, such as being restricted to systems with either shared or distributed memory or demanding the use of specialized compilers, operating systems, and libraries. Moreover, many of these do not handle fault tolerance transparently, an essential feature for scaling the execution of programs to a large number of processing nodes.

In this scenario, it is important to study the characteristics of parallelizable applications in order to take advantage of useful properties they might have. One such case is the class of *embarrassingly parallel problems*, which are applications that are trivially parallelized due to the fact that they can be immediately divided into completely independent parts that can be executed simultaneously [6]. Ray tracing in computer graphics [7], matrix multiplication [8], Monte Carlo integration [6], and several techniques to analyze seismic data [9]–[11] are important examples of such problems.

Designing and implementing parallel programs for embarrassingly parallel problems is usually no harder than doing so for their serial versions. Nonetheless, ensuring fault tolerant and efficient execution of these solutions on distributed systems is highly dependent on the programming model and runtime system used to create them. For instance, one could easily leverage the MPI standard to implement a parallel ray tracer by independently tracing the path of light through each of the image's pixels. However, ensuring that this program gracefully handles node failures on a large cluster would require a significant programming effort to include this mechanism. Also, guaranteeing performance scalability on systems that dynamically release or provide new computing resources during execution may be challenging when using MPI.

Several programming tools have emerged to aid the creation of scalable fault-tolerant systems. A few examples of this are Akka [12] for the actor model of concurrency in Java and Scala, Hadoop [13] for MapReduce computations, Pregel for graph processing [14], and TensorFlow [15] for machine learning. In each of their target applications, these engines improve productivity by letting the programmer focus on the task at hand instead of the minutia of the parallel execution.

The programming model used by the Scalable Partially Idempotent Tasks System (SPITS) [16] also abstracts execution details, thus allowing the easy development of distributed approaches for embarrassingly parallel problems. This programming model and its respective API provide a clear separation of the code created to solve the problem and the code required to distribute and manage the execution on parallel computing systems. As a result, the application developer can target the implementation of the solution while the runtime system developer can concentrate on how the

IEEE computer society

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:31:47 UTC from IEEE Xplore. Restrictions apply.

parallel execution will be performed.

In this study, we discuss how the properties of the SPITS programming model allow the development of simple runtime systems that tolerate faults and allow dynamic provisioning on large clusters. We also present a simple and effective reference runtime system implemented in Python and several experimental results that show that it is scalable and fault tolerant. Finally, we propose a performance model that can be used to aid users to understand when the performance of the runtime system scales with the number of computing nodes.

This paper is organized as follows: Section II presents related work. Section III discusses the programming model and how its properties simplify the implementation of fault tolerance and load balancing mechanisms. Section IV introduces the reference runtime system. Section V shows the experimental results. Finally, Section VI draws our conclusions.

II. Related Work

The *Task Superscalar* [17] programming model extracts task-level parallelism in the same way instruction-level parallelism is explored. A sequential thread dispatches tasks with its input/output properly annotated, allowing the runtime to construct a dependency graph, which is executed out-of-order by the available resources. Scheduling can be performed by either software [18]–[20] or hardware. Etsion et al. [17] argue that software solutions are inherently slow, impeding scalability. The advantage of this model is the simplicity to extract parallelism on complex problems: the programmer exposes which parameters are input, output or inout in the kernel functions and uses them as if the program was sequential.

Google engineers have faced this issue and discussed it on their MapReduce paper [21], where they proposed a new programming model to a) ease the implementation of programs to process large data sets and b) enable the implementation of a fault tolerant and scalable distributed runtime system for these programs. The motivation for their work lies on the observation that the developers that designed programs for straightforward computations were creating complex and obscure code to deal with runtime issues, such as fault tolerance and scalability. As a result, the MapReduce programming model enabled them to produce simple code that scaled gracefully on thousands of machines, even when computing nodes fail.

Similar to Google engineers, our group was producing a great amount of code to handle node failures when implementing MPI programs to solve highly parallel seismic processing problems. In order to reduce the programming complexity, we designed a new programming model and runtime architecture to: a) hide parallel execution details from the user, enabling them to focus on the problem rather than performance and scalability issues; and b) ensure minimal program properties that ease the implementation of fault tolerant and scalable runtime systems. We discuss this programming model and its system architecture in the following sections.

One of the main properties that our model relies on to simplify the fault tolerance and load balancing mechanisms of its runtime is idempotence [22], which has been used in many situations within the computer science field. De Kruijf and Sankaralingam [23] proposed an architecture of processors that leverage idempotence to implement speculative processor optimizations without using costly mechanisms of checkpointing to recover the architectural state. The authors observed that the same state could be rebuilt by simply re-executing idempotent regions. In another study by the same authors and Jha [24], they implemented a compilation technique aiming at dividing a program into idempotent regions. This time, the idempotence concept was extrapolated to tasks and this property was used to facilitate the implementation of fault tolerant mechanisms and load balancing in heterogeneous systems.

A second important feature for both our fault tolerance and load balancing solutions is the speculative execution of tasks, which is also a well-known approach. For instance, Alves et al. [25] introduce a Dataflow Error Recovery mechanism that adopts redundant execution for online error detection and recovery on dataflow systems and Marzulo et al. [26] employed speculation on the TALM model for dataflow execution to improve the system's performance.

III. SPITS Programming Model

When we first came across the challenge of scaling our programs to more than one machine, we decided to look past the MPI standard in order to seek models that would ease the implementation of fault tolerance and dynamic provisioning mechanisms. Despite the existence of several tools [5], [13], [18], most of them required specific versions of operating systems or the installation of software with root privileges, which was unfeasible for us. Another setback was the fact that some of the programming models relied on specialized compilers or complex runtime systems. As a result, we designed a simple programming model dedicated to our needs, which we called Scalable Partially Idempotent Tasks System (SPITS) [16].

Our initial attempt at designing this system resulted in a bag-of-tasks-based programming model that requires the user to provide only two functions: `generate_task` and `execute_task`. First, the runtime system repeatedly invokes `generate_task` on a master node to create new tasks, only stopping when an empty task is returned. Then, it spreads the tasks across the worker nodes. This is a simple action for the runtime system, as tasks are represented as a sequence of bytes. Finally, the runtime system executes the tasks in parallel by invoking `execute_task` on the Workers.

In this model, there is no restriction on tasks other than the fact that they must be serialized (encoded as sequence of bytes) when returned by `generate_task` and subsequently deserialized by `execute_task`. From a programmer's stand point, this interface has requirements, but is still straightforward, as it does not require handling parallel execution details such as scheduling and load balancing. Furthermore, this approach greatly simplifies the implementation from the perspective of the runtime system developer, seeing that tasks are merely arrays of bytes and can be easily transferred through the network or any persistent storage system.

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:31:47 UTC from IEEE Xplore. Restrictions apply.

Although this model allowed us to design uncomplicated runtime systems capable of both distributing tasks for parallel execution on clusters and managing the dynamic provisioning of resources, it was still not trivial to aggregate transparent handling of node failures to these structures. The main difficulty in implementing this feature being that we must make sure the partial execution does not affect the final result of the application in case a node fails during `execute_task`.

Checkpointing is a technique that can be used to guarantee that the partial execution does not affect the final results of the application. In this approach, the state of the application is periodically stored so that the last valid point in the execution can be recovered for restarting upon a failure. Despite being generic enough to address most of the problems brought by nodes failing, this method can also be very complex to implement on distributed runtime systems and may add high performance and memory overheads to the execution [27].

In our case, an alternative approach is to create the means for the `execute_task` function to be executed speculatively and its side effects only committed when the task has been carried out. This way, if there is an error while the function is running, the partial results are discarded and the task can be safely re-executed on another node. Nevertheless, ensuring that the speculative execution of the code is possible can prove to be difficult depending on the implementation of `execute_task`. Also, if an error happens during the commit operation, it may still compromise the state of the program.

To make the implementation of the fault tolerance mechanism easier for the runtime system developer, we propose that this burden be shared with the application programmer by requiring that the code have a property called *idempotence* [22].

A. Idempotence and Partially Idempotent Tasks

An idempotent operation is one that has the same effect whether you apply it once or more than once. For example, multiplying a number by zero is idempotent, since the result of $4 \times 0 \times 0 \times 0$ is the same as 4×0 [22].

Still, despite the fact that several executions of idempotent tasks always produce the same results, the consolidation of these outputs may not always be an idempotent operation in itself. Consider, for instance, a task that appends its results to the end of a file. Unlike file overwriting, appending is a non-idempotent operation, since if it were executed twice, the file would contain a duplicated task result.

When we analyze this, we see that requiring the function `execute_task` to be completely idempotent would greatly restrict the types of problems that can be solved with the aforementioned programming model. To address this issue, we proposed the concept of *partially idempotent tasks*, or PITs, where the computation of a task is subdivided in two phases: *execution* and *consolidation*; and, while the execution of the task is an idempotent operation, consolidation is not.

Compared to our first design, the new solution additionally requires that: a) the programmer make sure that `execute_task` is an idempotent function that takes a task as an argument and returns its results as an array of bytes; and b) they provide the `commit_task` function, responsible for consolidating the results produced by `execute_task`.

B. PITS, Fault Tolerance, and Load Balancing

In the execution phase, a runtime system for PITs can tolerate the failure of nodes that suddenly halt, stop receiving or sending messages, or take too long to respond by simply re-starting `execute_task` for the interrupted tasks on other nodes. We note that this approach does not detect these errors, only mitigates their effects through task speculation. Therefore, failures in which the node produces the wrong result are currently not handled.

As `generate_task` and `commit_task` are non-idempotent functions, they can not be re-executed in case of errors. Still, for our target applications (e.g., the seismic processing method CRS [9]), these functions are typically fast and can be efficiently executed by a small number of machines or perhaps even a single one, making failures very unlikely, as discussed by Dean and Ghemawat [21].

Another advantage of operating with tasks that have a guarantee of idempotence for part of their execution presents itself when we consider load balancing issues. Since the execution phase of a task can be carried out several times and always gives rise to the same final result, it is possible to speculatively schedule the `execute_task` function to run on idle nodes. In this way, computations of a certain task being performed on slower nodes can be interrupted when one of their copies finishes, thus better utilizing the resources of heterogeneous clusters and clouds.

IV. PY-PITS: an Open Source Reference Implementation

PY-PITS is an open source[1] reference implementation of the Scalable Partially Idempotent Tasks System (SPITS), which was proposed by Borin et al. [16]. It was created to allow further research and development of the API and the scheduling and resource managers. Most importantly, it also enabled the immediate development of commercial applications for our clients. We chose to implement the PY-PITS runtime in Python, since this language facilitates quick prototyping and development, does not require compilation, and is available on most UNIX-based systems. The current version of the runtime is compatible with Python 2.6, 2.7, and 3.x, which covers most of the versions shipped with older UNIX distributions.

Rather than implementing complex heuristics and algorithms to handle the parallel execution of several jobs at the same time, similar to the solutions adopted by TORQUE [28] and other resource managers, PY-PITS was designed to coordinate, instrument, and give detailed statistics on the execution of a single job. Nonetheless, more than one job can be concurrently executed in a cluster environment through multiple instances of PY-PITS. Additionally, PY-PITS is compatible with resource managers that were designed to work with MPI, allowing the deployment of the code on systems running legacy software and operating systems.

[1]https://github.com/eborin/pypits

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:31:47 UTC from IEEE Xplore. Restrictions apply.

A. *Architecture Overview*

Our runtime implementation follows the general architecture defined by the SPITS [16], where a Job Manager produces tasks that are consumed by Task Managers and dispatched to individual Workers. Once computed, the results are then sent to a Committer, which consolidates them. An optional main function may also be used to launch consecutive parallel segments of the same job instead of a single parallel job.

The PY-PITS runtime consists of two separate executables: a combined Job Manager/Committer and a Task Manager, responsible for loading the user module and handling the dynamic provisioning and fault tolerance mechanisms defined by the SPITS programming model.

B. *Combined Job Manager/Committer*

We decided to combine the Job Manager and the Committer into a single, multithreaded executable, since none of our target applications ever required the final result to be available on a node other than the one being used to generate tasks.

The management of tasks works by creating them on demand and appending them to the end of a circular queue that stores tasks already submitted to Task Managers. A task is then only removed from the queue once its result has been successfully received and committed. Once all tasks are created and submitted, the Job Manager assumes the ones which are still in the queue have failed, and starts re-submitting them to guarantee fault tolerance. The same mechanism allows the system to avoid long latencies on heterogeneous platforms.

The addition and removal of Task Managers is controlled by an announcing mechanism. Currently, the list of available nodes running Task Managers is provided through a file. Nodes listed in this file are accessed in a round-robin fashion by the Job Manager thread and tasks are sent to each Task Manager node until its internal queue is full. Similarly, the Committer thread accesses each node to fetch the completed tasks from its Task Manager's queue.

C. *Task Manager and Worker*

The PY-PITS Task Manager is responsible for consuming individual tasks and assigning them to Workers without any commitment to the order in which tasks were received. It leverages the dynamic loading mechanism available on GNU/Linux operating systems to load the user code so that Worker threads can run the `execute_task` function.

A multithreaded listener is responsible for serving requests from both the Job Manager and the Committer threads to maximize data throughput and overlap data transfers with task executions. Requests for pushing tasks do so through a task queue and the Task Manager can control an extra queue space to help hiding network latency. Requests for fetching results access a separate queue and return as many results as available.

A Worker is a theoretical construct from the SPITS model and, currently, it is translated directly into a thread in the PY-PITS runtime process. This design may cause the whole Task Manager to fail in case a Worker fails, but the system is still robust in the sense that unfinished tasks on this Task Manager will be re-executed by Workers on other Task Managers. Moreover, the dynamic provisioning system allows us to start another Task Manager to replace the one that failed. Since the SPITS model is sufficiently generic, sturdier mechanisms such as multiprocessing can be applied.

V. Experimental Results

We performed several experiments to validate PY-PITS. The results presented in this section were produced using a system with 3 nodes, each one configured with an Intel Xeon Ivy Bridge CPU E5-2640 v2 @ 2.60 GHz (6 cores/12 logical cores), 32 GB of RAM, a 2 TB HD, and Ubuntu 14.04 LTS. The nodes were connected through a Gigabit Ethernet Switch.

A. *Dynamic Provisioning and Fault Tolerance*

The dynamic provisioning test consists in progressively adding new nodes during the execution of a job, while the fault tolerance test is performed by terminating Task Manager processes during their execution. It is also possible to simulate an intermittent network connection by removing and adding nodes to the Job Manager/Committer, this latter test is particularly interesting because it shows that temporary connectivity losses do not prevent tasks from executing and being committed, as long as the connection between the Task Manager and the Committer becomes available again.

Figure 1 shows the commit rate when running a 180-task job. The x-axis indicates the time when a task is committed and the y-axis indicates the total number of tasks committed at that time. The size of the task is 256 kB and the computational time is 1 s. This test starts by having only one computing node with 1 Worker available to the Job Manager. Then, a second node with 1 Worker becomes accessible and, after that, a third one with 12 Workers is ready for use. Finally, the last node is disconnected to simulate a node failure. The first thing to notice is the effect in the committing rate when adding and removing nodes. The system commits roughly 1/2/14 tasks per second when working with 1/2/14 Workers. The second, and most important, observation is that the system was capable of executing and committing the 180 tasks even with a node being disconnected in the middle of the computation.

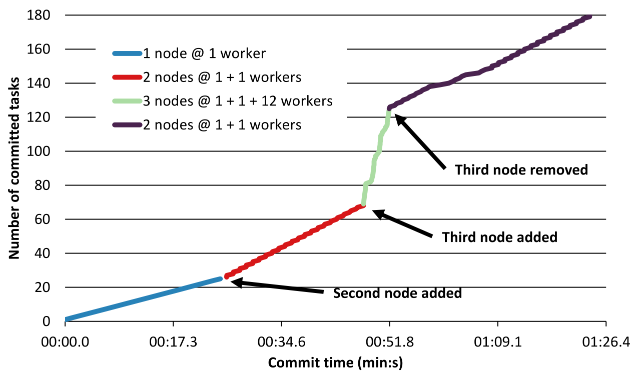

Fig. 1. Commit rate of tasks with dynamic provisioning.

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:31:47 UTC from IEEE Xplore. Restrictions apply.

The task submission policy also aims to solve heterogeneous performance issues since the runtime sees long running tasks as lost when the circular queue wraps around, causing them to be re-started on other nodes. If the original tasks commit before the newly-submitted ones, the later results are discarded.

B. Cluster Queues and Dynamic Provisioning

Computing clusters administrators usually implement job submission queues so users can share the system fairly. As an example, one of the clusters we share with other researchers at the university contains 32 nodes, with 40 cores each, and is configured with 3 submission queues:

1) small: allocates one node per job and allows the same user to run at most 20 jobs concurrently;
2) medium: allocates 2 to 4 nodes per job and allows the user to run at most 3 jobs concurrently;
3) large: allocates 5 to 8 nodes per job and does not allow the user to run jobs concurrently.

Curiously, there is no queue that allows the user to submit a job that uses the 32 nodes at the same time. One of the main reasons is that this kind of queue could cause the system to be underutilized. Underutilization happens because, before dispatching a job, the system must ensure that a minimum number of nodes is available, and in this case the system would wait for all running jobs to finish, possibly starving most of the nodes while some very long running jobs are running.

We created a small script that employs the dynamic provisioning mechanism implemented in PY-PITS to solve this problem. It checks for a file on the user directory that contains the address of the Job Manager. In case it does not exist, it creates a Job Manager to start the computation, otherwise, it creates a Task Manager and instructs it to connect to the already existing Job Manager. In this sense, the first job to run on the system creates the Job Manager, while the remaining jobs, which may be scheduled to run later, create new Task Managers that connect to the existing Job Manager and start computing tasks on its behalf. This approach allows us to fairly use all the 32 nodes of the system concurrently even if the queues do not allocate the whole cluster for a single job.

C. Bottleneck Analysis and Performance Model

In this section, we perform a bottleneck analysis and propose a performance model for the PY-PITS runtime. In order to evaluate the bottlenecks, we execute 1440 tasks using 2 Task Managers with 12 Workers each, a total of 24 Workers.

Defining the task length to be the amount of time the Worker takes to execute a task, we would expect the system to take 60 s $(1 \times (1440/24))$ to run 1440 tasks of length 1 s and 120 s to run 1440 tasks of length 2 s. In fact, this is what we see in Figure 2, which shows the time it takes PY-PITS to run 1440 tasks with 24 Workers under different task configurations.

Ideally, the performance of the system would be limited only by the number of Workers. However, in addition to executing the tasks, the runtime system must transfer the tasks from the Job Manager to the Task Managers and transfer their results back to the Committer, which takes time.

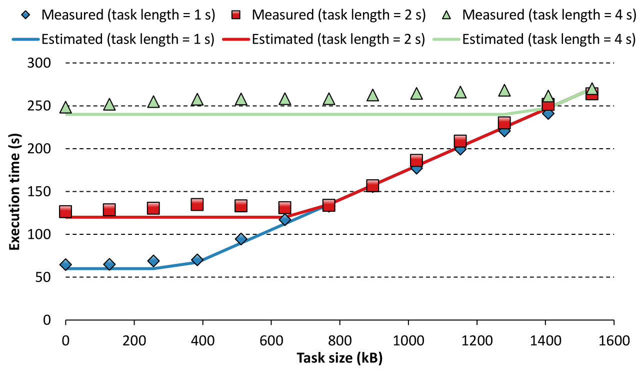

Fig. 2. Performance of PY-PITS running 1440 tasks with 24 Workers.

In order to hide the latency introduced by data transfers, PY-PITS buffer tasks and results so that data transfers can be overlapped with computation. In fact, as we can see in Figure 2, if we increase the size of tasks up to a given size there are no significant changes in execution time. Nonetheless, if the time it takes to transfer the tasks/results is longer than the time it takes to compute the tasks, the performance becomes limited by the network bandwidth instead of the number of Workers. This is the behavior we see when we increase the task size too much.

Since PY-PITS has only one Job Manager, we can model the performance of the system as:

$$\text{Exec. Time} = Max \begin{cases} N_{Tasks} \times (T_{len}/N_{Workers}) & (1) \\ N_{Tasks} \times T_{sz}/JM_{bw}) & (2) \end{cases}$$

where N_{Tasks} is the number of tasks, T_{len} is the time it takes to execute them (task length), T_{sz} is their size, $N_{Workers}$ is the number of Workers, and JM_{bw} is the Job Manager network bandwidth. Equation (1) estimates the time it would take to execute all tasks using $N_{Workers}$ and equation (2) estimates the time it would take to send all tasks from the Job Manager to the Task Managers. Since the execution of tasks is overlapped with data transfers, the time it takes to run the system is estimated as the maximum of (1) and (2). For the sake of simplicity we only include the Job Manager bandwidth on the model, however, in case the tasks results are larger than the tasks themselves, the model can be easily extended to take into account the bandwidth of the Committer.

As illustrated in Figure 2, the performance estimated by this model (indicated by solid lines) matches very well the performance observed in our experiments (indicated by markers).

By analyzing the performance model, we conclude that the performance only scales with the number of Workers in situations where:

$$T_{len} > \frac{T_{sz} \times N_{Workers}}{JM_{bw}}$$

Figure 3 shows the minimum task length to ensure the system performance scales with the number of Workers when $JM_{bw} = 8$ MB/s, which is the bandwidth measured during

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:31:47 UTC from IEEE Xplore. Restrictions apply.

our experiments. Note that, while 1 kB-tasks need to run for at least 1.53 s for the system performance to scale when executing with 100 000 nodes, the minimum task length requirement grows to 12 500 s for 8 MB-tasks.

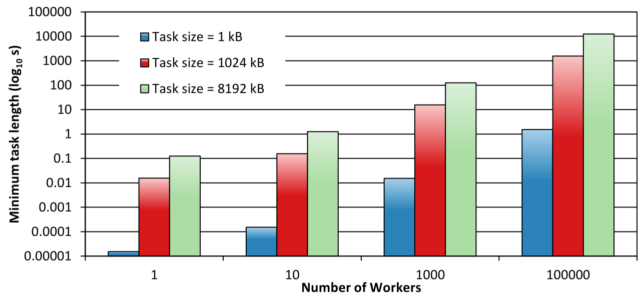

Fig. 3. Minimum task length to ensure the system scales with the number of Workers when $JM_{bw} = 8$ MB/s.

Typically, the tasks processed by our seismic processing programs have a few kilobytes and run for several seconds, hence, PY-PITS in its current state is capable of scaling the performance of our applications up to thousands of Workers. Nonetheless, in case one needs to run shorter (or larger) tasks on thousands of machines, the runtime system still has plenty of room for optimizations that would allow us to improve the results shown in Figure 3. As an example, the bandwidth measured in our experiments ($\sim$8 MB) is much lower than the performance that can be achieved on our hardware infrastructure.

VI. Conclusion

In this paper, we discussed how the SPITS programming model allowed us to design and implement PY-PITS, a simple and effective open source runtime system that tolerates faults and allows dynamic provisioning of resources during computation of PITs. We presented the design of PY-PITS and a few experimental results that showed that it is scalable and fault tolerant. Also, we discussed how PY-PITS can be used to improve utilization of multi-user computational cluster equipped with queues to submit jobs and, finally, we proposed a performance model to aid users to understand when the performance of PY-PITS scales with the number of Workers.

Future work includes the investigation of more fault-tolerance techniques and the execution of tests on a larger cluster in order to further explore the scalability of our system.

Acknowledgments

The authors thank Petrobras, CAPES, FAPESP, and CNPq for their invaluable financial support. We would also like to thank our colleagues at the HPG lab for their contributions.

References

[1] L. Dagum and R. Menon, “OpenMP: An Industry-Standard API for Shared-Memory Programming,” *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.

[2] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems,” *IEEE Comput. Sci. Eng.*, vol. 12, no. 3, pp. 66–73, May 2010.

[3] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable Parallel Programming with CUDA,” *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[4] J. Reinders, *Intel Threading Building Blocks*, 1st ed. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2007.

[5] M. P. I. Forum, “MPI: A Message-Passing Interface Standard,” University of Tennessee, Knoxville, TN, USA, Tech. Rep., Jun. 2015, accessed: August 18, 2016. [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[6] B. Wilkinson and M. Allen, *Parallel Programming: Techniques And Applications Using Networked Workstations And Parallel Computers*, 2nd ed. New York, NY, USA: Pearson Education, 2006.

[7] A. Chalmers, T. Davis, and E. Reinhard, *Practical Parallel Rendering*, 1st ed. Natick, MA, USA: A. K. Peters, Ltd./CRC Press, 2002.

[8] N. Matloff, *Parallel Computing for Data Science: With Examples in R, C++ and CUDA*, 1st ed. Boca Raton, FL, USA: CRC Press, 2015.

[9] R. Jäger, J. Mann, G. Höcht, and P. Hubral, “Common-reflection-surface stack: Image and attributes,” *Geophysics*, vol. 66, no. 1, pp. 97–109, Jan. 2001.

[10] O. C. Reyes, J. D. L. Puente, V. Puzyrev, and J. M. Cela, “Parallel and numerical issues of the edge finite element method for 3D controlled-source electromagnetic surveys,” in *Proc. ICCSAT’15*, 2015, pp. 1–6.

[11] M. Hori, T. Ichimura, M. L. L. Wijerathne, and K. Fujita, *Application of HPC to Earthquake Hazard and Disaster Estimation*. Springer International Publishing, Nov. 2014, pp. 203–220.

[12] J. Goodwin, *Learning Akka*, 1st ed. Birmingham, Warks, UK: Packt Publishing, 2015.

[13] T. White, *Hadoop: The Definitive Guide*, 3rd ed. Sebastopol, CA, USA: O’Reilly Media, 2012.

[14] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: A System for Large-scale Graph Processing,” in *Proc. SIGMOD’10*, 2010, pp. 135–146.

[15] M. Abadi *et al.*, “TensorFlow: A system for large-scale machine learning,” Google Brain, Mountain View, CA, USA, Tech. Rep., May 2016, accessed: August 19, 2016. [Online]. Available: https://arxiv.org/abs/1605.08695

[16] E. Borin, I. L. Rodrigues, A. T. Novo, J. D. Sacramento, M. Breternitz, and M. Tygel, “Efficient and Fault Tolerant Computation of Partially Idempotent Tasks,” in *Proc. SBGf’15*, 2015, pp. 367–372.

[17] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, “Task Superscalar: An Out-of-Order Task Pipeline,” in *Proc. MICRO’10*, 2010, pp. 89–100.

[18] J. Labarta, “StarSS: A programming model for the multicore era,” in *PRACE WorkshopNew Languages & Future Technology Prototypes at the Leibniz Supercomputing Centre in Garching (Germany)*, 2010.

[19] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, *An Extension of the StarSs Programming Model for Platforms with Multiple GPUs*. Springer Berlin Heidelberg, Aug. 2009, pp. 851–862.

[20] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, “CellSs: a Programming Model for the Cell BE Architecture,” in *Proc. SC’06*, 2006, pp. 5–5.

[21] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[22] S. R. Leonard Richardson, *RESTful Web Services*, 1st ed. Sebastopol, CA, USA: O’Reilly Media, 2008.

[23] M. de Kruijf and K. Sankaralingam, “Idempotent Processor Architecture,” in *Proc. MICRO’11*, 2011, pp. 140–151.

[24] M. A. de Kruijf, K. Sankaralingam, and S. Jha, “Static Analysis and Compiler Design for Idempotent Processing,” in *Proc. PLDI’12*, 2012, pp. 475–486.

[25] T. A. O. Alves, S. Kundu, L. A. J. Marzulo, and F. M. G. França, “Online error detection and recovery in dataflow execution,” in *Proc. IOLTS’14*, July 2014, pp. 9–104.

[26] L. A. J. Marzulo, T. A. O. Alves, F. M. G. França, and V. S. Costa, “TALM: A Hybrid Execution Model with Distributed Speculation Support,” in *Proc. SBAC-PADW’10*, Oct 2010, pp. 31–36.

[27] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, “A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems,” *J. Supercomput.*, vol. 65, no. 3, pp. 1302–1326, Sep. 2013.

[28] G. Staples, “TORQUE Resource Manager,” in *Proc. SC’06*, 2006, article no. 8.

Authorized licensed use limited to: b-on: Universidade de Lisboa Reitoria. Downloaded on May 13,2020 at 15:31:47 UTC from IEEE Xplore. Restrictions apply.